

```

// Computer Program Listing Appendix Under 37 CFR 1.52(e)
/*
Code samples:
C++ source (inside the engine):
gen.cpp - routines for generating columns of virtual indexes given equality and
inequality predicates (i.e. from the Generator, 410)
gen2.cpp - routines for controlling the optimization and recommendation of a set
of queries in the engine (i.e. from the IXT unit, 430)
Java classes (on the client side, in the Driver unit, 470):
IxtColumn.java - class encapsulating a column (of a table or index)
IxtConfiguration.java - class representing a binding between a query and a set
of virtual indexes picked by the optimizer for a query
IxtDriver.java - main class that controls the execution of the index
recommendation process
IxtElement.java - class encapsulating an index element; that is, a column of an
index, together with it's position and sortedness
IxtIndex.java - class encapsulating an index, either physical or virtual
IxtInstance.java - class encapsulating an instance of the problem the index
consultant is analyzing; contains the catalog information
from the database and the queries from the workload
IxtPhase.java - class encapsulating all virtual indexes and cost information
associated with a single phase
IxtQuery.java - class encapsulating a query from the workload
IxtTable.java - class encapsulating a base table in the database
*/

// gen.cpp
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*****
 * Index Generator code      *
 *****/

/* Simple bit-vector manipulation */
#define _is_marked( vec, pos ) ( ( vec >> pos ) & 0x00000001 )
//must be smaller than MAX_VIRTUAL_IDX_COLUMNS
#define NUM_PREDS_TO_CONSIDER 5
ixt_Generator::ixt_Generator( ixt_IndexTuningManager *itm,
                             qog_Quantifier *q )
: _itm( itm ),
  _quantifier( q )
/*****/

/* The ixt_Generator will create virtual indexes for all relevant
 * combinations of columns in quantifier q. Relevant columns are
 * contained in predicates and order-by constraints.
 */
{
}

ixt_Generator::~ixt_Generator()
/*****/
{
}

void

```

```

ixt_Generator::Generate( qog_OrderProperty * io )
/*****
/* For each existing virtual index, try extending it with the current
* interesting order.
*
* -Do safety checks and avoid generation on temporary or system tables
* -For each virtual index on the table in question:
* -If the index is not full and has not yet been extended, attempt
* to extend it with io
* -Extend an empty index with the current interesting property
*
*/
{
    ixt_TableWrapper *tab;
    p_TableDef td = _quantifier->GetBaseTableDefinition();
    // Safety check to prevent crashing
    if( td == NULL )
    {
        _assertD( FALSE );
        return;
    }
    // Skip generation on temp/system tables
    if( _is_temp_table( td ) || td->is_catalog_table() ){
        return;
    }
    tab = _itm->FindOrAddTableWrapperByID( td->GetTableID() );
    ixt_VirtualIndexInfo *vii;
    if( tab == NULL )
    {
        return;
    }
    tab->InitSimpleIndexIterator();
    vii = tab->GetNextIndex();
    for( vii = tab->GetNextIndex(); vii != NULL; vii = tab->GetNextIndex() ) {
        if( !vii->enabled ) {
            continue;
        }
        if( vii->hardened < MAX_VIRTUAL_IDX_COLS - 1
            && vii->cols[vii->hardened] != NULL ) {
            ExtendWithOrder( vii, io );
        }
    }
    ExtendWithOrder( NULL, io );
}
void
ixt_Generator::ExtendWithOrder( ixt_VirtualIndexInfo * vii,
    qog_OrderProperty * io )
/*****
/* Add columns to virtual index vii to give it the specified order property.
* If some column from the order property already exist in the index, rearrange

```

```

* them as necessary to match the order property. In the worst case, the
* columns from the order property will simply be appended to the existing
* columns.
*/
{
    p_expr expr;
    a_bool is_dontcare;
    a_bool matched;
    int i;
    int j;
    int numcols = 0;
    ixt_VirtualIndexInfo *new_vii;
    p_TableDef td = _quantifier->GetBaseTableDefinition();
    if( vii != NULL ) {
        if( vii->hardened == MAX_VIRTUAL_IDX_COLS
            || vii->cols[vii->hardened] != NULL ) {
            //we can't extend an index that has already been extended
            //(ie. has hardened columns)
            return;
        } else {
            numcols = vii->hardened;
        }
    }
    new_vii = _itm->AddVirtualIndex( td->GetTableID(), NULL, io );
    if( new_vii == NULL ) {
        _assertD( FALSE );
    }
    return;
}
for( i = 0; i < numcols; i++ )
{
    new_vii->cols[i] = vii->cols[i];
    new_vii->seq[i] = vii->seq[i];
}
for( i = 0; i < io->Size() && i < MAX_VIRTUAL_IDX_COLS; i++ )
{
    matched = FALSE;
    expr = io->ElementExpr( i );
    is_dontcare = ! io->ElementFixedPosition( i );
    for( j = 0; vii != NULL && j < vii->hardened; j++ )
    {
        if( new_vii->cols[j] == expr->GetBaseColumn() )
        {
            matched = TRUE;
        }
    }
    if( !matched )
    {
        _assertD( expr->GetBaseColumn() != 0 );
        new_vii->cols[i] = expr->GetBaseColumn();
        if( is_dontcare )

```

```

    {
new_vii->hardened++;
new_vii->seq[i] = INSENSITIVE_ORDER;
    } else {
new_vii->seq[i] = io->ElementSequence(i);
    }
    numcols++;
}
}
if( _itm->IsUseClustered() ) {
new_vii->is_clustered = TRUE;
}
ixt_VirtualArrayIndex vai(new_vii);
vai.EstimateSize();
if( vii != NULL ) {
a_bool is_exact_duplicate = TRUE;
if( new_vii->hardened != vii->hardened ) {
    is_exact_duplicate = FALSE;
}
for( i = new_vii->hardened; i < MAX_VIRTUAL_IDX_COLS; i++ ) {
    if( vii->cols[i] != new_vii->cols[i] ) {
is_exact_duplicate = FALSE;
    }
}
}
if( is_exact_duplicate ) {
    // this prevents us from entering an infinte loop
    new_vii->valid = FALSE;
}
}
}
void
ixt_Generator::ExtendWithInequalities( ixt_VirtualIndexInfo *vii,
    volcano_vector_subset **neqs )
/*****
/* Add a column to the virtual index vii if necessary
*
* -For each column in the table:
* -If there is a sargable inequality predicate on the column
*   -Create a new virtual index structure new_vii
*   -Copy the columns from vii to new_vii
*   -Append the current column to the index
*   -Initialize the costing information for the index
*/
{
    int i;
    int j;
    int numcols = 0;
    ixt_VirtualIndexInfo *new_vii;
    p_TableDef td = _quantifier->GetBaseTableDefinition();
    if( vii != NULL ) {

```

```

numcols = vii->hardened;
}
for( i = 0; i < td->num_cols; i++ ) {
if( neqs[i] != NULL ){
    new_vii = _itm->AddVirtualIndex( td->GetTableID(),
        NULL,
        _quantifier );
    if( new_vii == NULL ) {
_assertD( FALSE );
return;
    }
    for( j = 0; j < numcols; j++ )
    {
new_vii->cols[j] = vii->cols[j];
new_vii->seq[j] = vii->seq[j];
    }
    new_vii->cols[j] = td->FindColumnByIndex(i);
    new_vii->seq[j] = INSENSITIVE_ORDER;
    new_vii->hardened++;
    if( _itm->IsUseClustered() ) {
new_vii->is_clustered = TRUE;
    }
    {
ixt_VirtualArrayIndex vai(new_vii);
vai.EstimateSize();
    }
}
}
}

typedef struct colinfo {
    a_selectivity sel;
    p_column_def cd;
} a_colinfo;

void
ixt_Generator::Generate( volcano_vector_subset **eq_preds,
    volcano_vector_subset **neq_preds )
/*
/*****
/* Generate a set of indexes for each combination of eq preds (where all of the
/* columns of such indexes will be don't care with respect to both position and
/* sortedness. Also, for each such index, copy it and extend it with each of the
/* inequality predicates.
*/
{
    int i;
    int j;
    int k;
    p_TableDef td = _quantifier->GetBaseTableDefinition();
    a_colinfo cols[NUM_PREDS_TO_CONSIDER];
    int numcols = 0;
    int to_add = 0;

```

```

a_colinfo cur;
dfp *pred;
ixt_VirtualIndexInfo *vii;
//don't suggest indexes on temp tables or system tables
if( _is_temp_table( td ) || td->is_catalog_table() ){
return;
}
cur.sel = 1;
cur.cd = NULL;
for( i = 0; i < NUM_PREDS_TO_CONSIDER; i++ )
{
cols[i] = cur;
}
if( eq_preds != NULL ){
for( i = 0; i < td->num_cols; i++){
if( eq_preds[i] != NULL ){
cur.cd = td->FindColumnByIndex(i);
cur.sel = 1;
volcano_vector_subset_iter myiter( eq_preds[i] );
myiter.Establish( eq_preds[i] );
for( pred = (dfp*)myiter.GetNextObject();
pred != NULL;
pred = (dfp*)myiter.GetNextObject() ) {
cur.sel *= pred->GetSelectivity();
}
}
for( j = 0; j < NUM_PREDS_TO_CONSIDER; j++ ) {
if( cur.sel < cols[j].sel ){
// move all the lower predicates down the list
for( k = NUM_PREDS_TO_CONSIDER - 1; k > j; k-- ) {
cols[k] = cols[k-1];
}
cols[j] = cur;
numcols++;
break;
}
}
}
}
numcols = _min( numcols, NUM_PREDS_TO_CONSIDER );
for( i = 1; i < (1 << numcols); i++ ) {
vii = _itm->AddVirtualIndex( td->GetTableID(), NULL, _quantifier );
if( vii == NULL ){
_assertD( FALSE );
return;
}
}
to_add = 0;
for( j = 0; j < NUM_PREDS_TO_CONSIDER; j++ ) {
if( _is_marked( i, j ) ){
vii->cols[to_add] = cols[j].cd;

```

```

vii->seq[to_add] = INSENSITIVE_ORDER;
to_add++;
}
}
if( _itm->IsUseClustered() ){
    vii->is_clustered = TRUE;
}
vii->valid = TRUE;
{
    ixt_VirtualArrayIndex vai( vii );
    vai.EstimateSize();
    vai.SetFirstHardened( to_add );
}
ExtendWithInequalities( vii, neq_preds );
}
// to generate NEQ-only indexes
ExtendWithInequalities( NULL, neq_preds );
}
/*****
 * Qoq Visitor code
 *****/
static void
back_propagate_order( ixt_VirtualIndexInfo * vii, qoq_OrderProperty * op )
/*****/
/* Make a virtual index have the same ordering specified in the
 * qoq_OrderProperty. This may require rearranging some don't care columns
 * and fixing them in place.
 *
 */
{
    p_column_def cols[MAX_VIRTUAL_IDX_COLS];
    SORT_SEQUENCE seq[MAX_VIRTUAL_IDX_COLS];
    int i, j;
    // Clear the result
    for( i=0; i<MAX_VIRTUAL_IDX_COLS; ++i ) {
        cols[i] = NULL;
        seq[i] = INSENSITIVE_ORDER;
    }
    vii->hardened = MAX_VIRTUAL_IDX_COLS;
    // Place all columns that are missing from the final order
    // property at their original position
    for( i=0; i<MAX_VIRTUAL_IDX_COLS && vii->cols[i] != NULL; ++i ) {
        // Try to find the column in the order property
        a_bool found = FALSE;
        for( j=0; j<op->Size(); ++j ) {
            if( vii->cols[i] == op->ElementExpr(j)->GetBaseExpr()->GetBaseColumn() ) {
                found = TRUE;
                break;
            }
        }
    }
}

```

```

// If not found, place the column at the original position
if( !found ) {
    cols[i] = vii->cols[i];
    seq[i] = vii->seq[i];
    if( seq[i] != INSENSITIVE_ORDER && i < vii->hardened ) {
        vii->hardened = i;
    }
}
}
// Fill in remaining columns in the same order they appear in the
// final order property
i = 0;
for( j=0; j<op->Size(); ++j ) {
    // Find next unfilled position in the result
    while( cols[i] != NULL ) {
        _assertD( i < MAX_VIRTUAL_IDX_COLS );
        ++i;
    }
    cols[i] = op->ElementExpr(j)->GetBaseExpr()->GetBaseColumn();
    seq[i] = op->ElementSequence(j);
    if( op->ElementFixedPosition(j) && i < vii->hardened ) {
        vii->hardened = i;
    }
}
// Copy the result back to the vii
for( i=0; i<MAX_VIRTUAL_IDX_COLS && vii->cols[i] != NULL; ++i ) {
    vii->cols[i] = cols[i];
    vii->seq[i] = seq[i];
}
if( i < vii->hardened ) {
    vii->hardened = i;
}
}
// gen2.cpp
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
a_bool
ixt_IndexTuningManager::runQuery( p_expr expr )
/*****
/* Perform index tuning for the query described in the string expr. This
* requires that _master_id and _phase have already been appropriately
* set. If this is phase 1, we will optimize the query twice: once in the
* 'vanilla' state, to provide a baseline for future comparisons, and once
* in the . The vanilla state will always include indexes that satisfy RI
* constraints. In addition, if the keepExisting option has been set,
* the baseline also includes all other physical indexes, since we are
* only interested in the incremental benefit.
*/
{
    a_heap_ref stmtheap;
    SQLAHeap heap(1);

```

```

a_statement * stmt;
p_db_cursor gen_crsr = NULL;
p_db_cursor vanilla_crsr = NULL;
uint32 hash_value = 0;
a_bool is_new = TRUE;
an_estimate_cost estimated_total = 0;
a_bool ret = TRUE;
ixt_IndexTuningTables *itt = GetTuningTables();
_is_gen_run = FALSE;
stmtheap.mem = PrepareExpr( expr, GOAL_STATEMENT, NULL, FALSE );
heap.HeapSet( stmtheap.mem );
stmt = (a_statement *)stmtheap.mem;
if( stmt == NULL
|| stmt->ptr == NULL
|| stmt->type != STMT_INSERT
&& stmt->type != STMT_SELECT
&& stmt->type != STMT_UPDATE
&& stmt->type != STMT_DELETE ){
WriteErrorToIx_ConsultantLog( itt, _workload_id, _phase );
return FALSE;
}
if( _phase == 1 ) {
ixt_ParseTreeHashVisitor h( NULL );
hash_value = h.CheckParseTree( stmt );
_query_text_id = itt->FindOrAddQueryText( hash_value,
stmt->type,
&is_new );
if( !is_new ){
//we have already seen a variant of this query - don't execute
//it again
goto cleanup;
}
//we don't run to run statements that return a plan
if( stmt->type == STMT_SELECT
&& ( _strieq(stmt->strings->buff,"plan")
|| _strieq(stmt->strings->buff,"graphical_plan")
|| _strieq(stmt->strings->buff,"explanation")
|| _strieq(stmt->strings->buff,"graphical_ulplan")
|| _strieq(stmt->strings->buff,"short_ulplan")
|| _strieq(stmt->strings->buff,"long_ulplan") ) )
{
ret = FALSE;
goto cleanup;
}
vanilla_crsr = BuildCursor( stmt );
if( vanilla_crsr == NULL ){
WriteErrorToIx_ConsultantLog( itt, _workload_id, _phase );
goto cleanup;
} else {
if( vanilla_crsr->GetSimpleQueryCursor() == NULL

```

```

&& ( vanilla_crsr->GetQog() == NULL
    || !vanilla_crsr->GetQog()->Optimize() ) )
{
//this query is not a simple cursor and could not be
//optimized
WriteErrorToIx_ConstantLog( itt, _workload_id, _phase );
goto cleanup;
} else {
_debug(
    itt->AddLogMessage( _phase,
        IX_CONSULTANT_LOG_DEBUG,
        _workload_id,
        0,
        "Costed vanilla query successfully!" );
    )
}
}
uint32 numrows = 0;
if( vanilla_crsr->GetQog() != NULL ){
    if( stmt->type == STMT_DELETE
    || stmt->type == STMT_INSERT
    || stmt->type == STMT_UPDATE )
    {
        numrows = vanilla_crsr->GetQog()->GetRootDTB()->GetDTBCost()->GetNumRows();
    }
    estimated_total = getCostFromCursor( vanilla_crsr );
} else {
    //should only happen for INSERT VALUES statements
    numrows = 1;
    estimated_total = 0;
    _assertD( stmt->type == STMT_INSERT );
}
itt->UpdateQueryTextCost( _query_text_id,
    estimated_total,
    numrows );
_is_gen_run = TRUE;
}
gen_crsr = BuildCursor(stmt);
if( gen_crsr == NULL ){
WriteErrorToIx_ConstantLog( itt, _workload_id, _phase );
ret = FALSE;
goto cleanup;
} else {
if( gen_crsr->GetQog() == NULL
    || !gen_crsr->GetQog()->Optimize() ){
    WriteErrorToIx_ConstantLog( itt, _workload_id, _phase );
    ret = FALSE;
    goto cleanup;
} else {
    _debug(

```

```

itt->AddLogMessage( _phase,
    IX_CONSULTANT_LOG_DEBUG,
    _workload_id,
    0,
    "Costed generated query successfully!" );
)
estimated_total = getCostFromCursor(gen_csr);
itt->FindOrAddQueryPhase( _phase, _query_text_id, getCostFromCursor( gen_csr ) );
}
}
if( vanilla_csr != NULL ){
ixt_QueryStatGatherer * gath = new ixt_QueryStatGatherer( this );
ixt_QogVanillaVisitor vis( this, gath );
vis.Visit_QOG( vanilla_csr->GetQog() );
gath->PrintToLog();
delete gath;
}
if( gen_csr != NULL ){
ixt_QogIndexVisitor vis( this, NULL );
vis.Visit_QOG( gen_csr->GetQog() );
}
cleanup:
_is_gen_run = FALSE;
if( vanilla_csr != NULL ){
vanilla_csr->fini_qog();
DB_Fini_db_cursor( vanilla_csr );
}
if( gen_csr != NULL ){
ResetIndexTuning();
gen_csr->fini_qog();
DB_Fini_db_cursor( gen_csr );
}
DV_Free_heap( &stmtheap );
return ret;
}

void
ixt_IndexTuningManager::RecommendIndexes( uint32 master_id,
    uint32 phase,
    a_bool use_clustered,
    a_bool keep_existing )
/*****
/* Recommend indexes for all queries stored in the workload_table. The method
* loads the queries, sets the context for each query (ie. user, option
* settings), then calls the method to recommend indexes for an individual
* query. At the end, the code cleans up any changes it made to the current
* connection.
*/
{
    p_table_def workload_table;
    p_column_def query_col;

```

```

p_column_def user_col;
p_column_def workload_id_col;
p_column_def text_id_col;
p_column_def discarded_col;
p_column_def cache_size_col;
p_column_def opt_goal_col;
p_column_def opt_level_col;
p_column_def user_estimates_col;
p_column_def plan_hash_col;
p_expr cache_size_expr;
p_expr opt_goal_expr;
p_expr opt_level_expr;
p_expr user_estimates_expr;
p_expr plan_hash_expr;
p_expr new_query_expr;
p_expr user_expr;
p_expr text_id_expr;
p_expr workload_id_expr;
p_expr discarded_expr;
a_SimpleQueryCursor * c;
ixt_VirtualArrayIndex current( NULL );
UserDef *user;
char buff[100];
a_bool successful_run;
int num = 0;
p_Connection conn = _CurrentConnection;
if( !conn->db()->has_index_tuning_tables() ){
return;
}
if( _ix_consultant_tables.GetTables() < 5 ){
_ix_consultant_tables.ReleaseTables();
_assertD( FALSE );
return;
}
//Turn on automatic index tuning mode
StartIndexTuning();
//Initialize flags
_master_id = master_id;
_phase = phase;
_keep_existing = keep_existing;
_use_clustered = use_clustered;
_query_text_id = 0xffffffff;
workload_table = _ix_consultant_tables.GetTableDef(IX_CONSULTANT_WORKLOAD);
_assertD( workload_table != NULL );
if( workload_table == NULL ) return;
query_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__TEXT + 1 );
user_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__USER_ID + 1 );
workload_id_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__WORKLOAD_ID + 1 );
text_id_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__TEXT_ID + 1 );
discarded_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__DISCARDED + 1 );

```

```

cache_size_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__CACHE_SIZE + 1 );
opt_goal_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__OPTIMIZATION_GOAL + 1
);
opt_level_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__OPTIMIZATION_LEVEL + 1
);
user_estimates_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__USER_ESTIMATES
+ 1 );
plan_hash_col = workload_table->FindColumnByID( IX_CONSULTANT_WORKLOAD__PLAN_HASH + 1 );
c = SC_New( NULL, workload_table, NULL, phase > 1 ? 1 : 0 );
new_query_expr = c->RequireColumnDef( query_col );
user_expr = c->RequireColumnDef( user_col );
workload_id_expr = c->RequireColumnDef( workload_id_col );
text_id_expr = c->RequireColumnDef( text_id_col );
discarded_expr = c->RequireColumnDef( discarded_col );
cache_size_expr = c->RequireColumnDef( cache_size_col );
opt_goal_expr = c->RequireColumnDef( opt_goal_col );
opt_level_expr = c->RequireColumnDef( opt_level_col );
user_estimates_expr = c->RequireColumnDef( user_estimates_col );
plan_hash_expr = c->RequireColumnDef( plan_hash_col );
if( phase > 1 ){
c->OrderByColumn( IX_CONSULTANT_WORKLOAD__TEXT_ID, TRUE );
}
c->SearchNum( IX_CONSULTANT_WORKLOAD__MASTER_ID, _master_id );
while( c->Fetch() ) {
num++;
if( phase > 1 && text_id_expr->v.ul == _query_text_id ){
// we only want to execute one of each query type
if( num % 10 == 0 ) {
SendMarker( num );
}
continue;
}
SendMarker( num );
if( !discarded_expr->is_null ){
// we are not considering this query because it is either too
// cheap to be relevant, or it is poorly-behaved
continue;
}
user = FindUserByID( user_expr->v.ul );
setQueryParams( cache_size_expr,
opt_goal_expr,
opt_level_expr,
user_estimates_expr,
user );
if( user != NULL ){
conn->SetUser( user->GetSAUserName(), TRUE, TRUE );
user->Release();
} else {
_assertD(FALSE);
}
}

```

```

_workload_id = workload_id_expr->v.ul;
_query_text_id = text_id_expr->v.ul;
successful_run = runQuery( new_query_expr );
if( successful_run && phase == 1 ){
    c->DefineNum( IX_CONSULTANT_WORKLOAD__TEXT_ID, _query_text_id );
    c->ModifyCurrent();
} else if( !successful_run ) {
    // Clear the error - otherwise we can't flag the query as
    // discarded. Error should have been logged inside runQuery().
    if( SQLErr( conn ) ) {
        dbi_reset_error();
    }
    //flag this as a discarded query
    c->DefineNum( IX_CONSULTANT_WORKLOAD__TEXT_ID, _query_text_id );
    c->ModifyCurrent();
    c->DefineNum( IX_CONSULTANT_WORKLOAD__DISCARDED, 1 );
    c->ModifyCurrent();
    // WriteErrorToIx_ConstantLog( &_ix_consultant_tables, _workload_id, _phase );
}
if( SQLErr( conn ) ) {
    dbi_reset_error();
}
if( Context_switch( TRUE ) ) {
    _CurrentConnection->DBLangString().FormatMiscLangStr( IDS_INTERRUPTED, buf, sizeof(buf) );
    _ix_consultant_tables.AddLogMessage( phase,
        IX_CONSULTANT_LOG_ERR,
        _workload_id,
        SQLSTATE_INTERRUPTED,
        buf );
    sql_error( SQLSTATE_INTERRUPTED );
    break;
}
}
if( !SQLErr() ){
_ix_consultant_tables.AddLogMessage( phase,
    IX_CONSULTANT_LOG_INFO,
    _workload_id,
    0,
    "Completed phase" );
}
ResetIndexTuning( TRUE );
delete c;
c = NULL;
conn->SetUser( NULL, TRUE, TRUE );
_ix_consultant_tables.ReleaseTables();
FiniIndexTuning();
}
void
ixt_QogIndexVisitor::Visit_qog_RelevantIndex( qog_RelevantIndex * node )
/*****

```

```

/* Record the information about a given index in the *_ix_consultant_ixcol table.
*/
{
    lIndex *idx = node->GetIndex();
    _assertD( idx != NULL );
    SORT_SEQUENCE seq;
    ixt_VirtualArrayIndex *vai = NULL;
    int i;
    an_estimate_cost local_score = 0.0;
    uint32 phase = _itm->GetPhase();
    an_estimate_row_count cluster_score = 0;
    qog_OrderProperty *op = node->APGetOrderProperty()->GetDep();
    if( idx != NULL
    && !idx->IsPrimaryKey()
    && !idx->IsForeignKey() )
    {
        cluster_score = node->APGetCost()->GetReadCost();
        local_score = node->GetQuantifier()->APGetCost()->GetTotalCost();
        _index_id = idx->GetIndexID();
        _itm->GetTuningTables()->FindOrAddIndex( phase,
            _table_id,
            _index_id,
            idx->GetLeafPageCount(),
            idx->IsClustered(),
            idx->IsVirtual() );
        _itm->GetTuningTables()->AddQueryIndexPair( phase,
            _query_text_id,
            _pos,
            _table_id,
            _index_id,
            local_score,
            cluster_score );
        if( idx->IsVirtual() ){
            vai = (ixt_VirtualArrayIndex *)idx;
            if( !vai->GetHasBeenDumped() ){
                vai->SetHasBeenDumped();
            } else {
                return;
            }
            if( op != NULL ) {
                back_propagate_order( vai->GetInfo(), op );
            }
        }
    }
    for( i = 0; i < idx->GetNumColsIndexed(); i++){
        if( idx->IsVirtual() ){
            seq = vai->GetSortSequence(i);
            _itm->GetTuningTables()->AddIndexColumn( phase,
                _table_id,
                _index_id,
                i,

```

```

        idx->GetColumn( i )->GetColumnID(),
        seq );
    }
}
}
}
// IxtColumn.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****
package com.sybase.indexConsultant;
public class IxtColumn
{
    long _id;
    String _name;
    IxtTable _table;
    public IxtColumn( long id, String name, IxtTable table )
    {
        _id = id;
        _name = name;
        _table = table;
    }
    public
    long getID()
    {
        return _id;
    }
    public
    String getName()
    {
        return _name;
    }
    public
    IxtTable getTable()
    {
        return _table;
    }
}
// IxtConfiguration.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****
package com.sybase.indexConsultant;
import java.util.*;
public class IxtConfiguration
{
    IxtQuery __query;
    Vector __indexes;

```

```

    double    _cost;
    double    _workingcost;
    public IxtConfiguration( IxtQuery query, double cost, Vector indexes )
    {
        _query = query;
        _indexes = indexes;
        _cost = cost;
        _workingcost = cost;
    }
    public
    int currentCostCompare( IxtConfiguration other )
    {
        return Double.compare( _workingcost, other._workingcost );
    }
    public
    Iterator getIndexIterator()
    {
        return _indexes.iterator();
    }
    public
    double getWorkingCost()
    {
        return _workingcost;
    }
    public
    double getRealCost()
    {
        return _cost;
    }
    public
    double getBenefit()
    {
        return ( _query.getVanillaCost() - _workingcost ) * _query.getCount();
    }
    public
    void removeIndex( IxtIndex ix )
    {
        _indexes.remove( ix );
    }
    public
    void addIndex( IxtIndex ix )
    {
        _indexes.add( ix );
    }
    public
    IxtQuery getQuery()
    {
        return _query;
    }
}

```

```

// IxtDriver.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.sql.*;
import java.util.*;
public class IxtDriver
{
    static final double PHASE_REDUCTION = 0.2;
    static IxtDriver _global;
    Statement _stmt;
    IxtAnalysis _analysis;
    IxtPhase _curphase;
    boolean _clustered_option;
    boolean _keep_existing_option;
    long _size_constraint;
    long _master_id;
    boolean _in_progress;
    boolean _is_done;
    boolean _can_stop_early;
    boolean _cancelTuning;
    // Feedback control to send information back to the caller of IxtDriver
    IxtFeedback _feedbackControl = null;
    static public
    boolean getClusteredOption()
    {
return _global._clustered_option;
    }
    static public
    boolean getKeepExistingOption()
    {
return _global._keep_existing_option;
    }
    static public
    long getMasterID()
    {
return _global._master_id;
    }
    static public
    long getSizeConstraint()
    {
return _global._size_constraint;
    }
    static public
    void showSubstatus( String message )
    {
        _global._feedbackControl.showSubstatus( message );
    }
}

```

```

static public
void showStatus( String message )
{
_global._feedbackControl.showStatus( message );
}
static public
lxtPhase getCurrentPhase()
{
return _global._curphase;
}
private
int getSizeOfUselessPhysicalIndexes()
/* Add up the size of all physical indexes that have not been used by the
* optimizer to answer queries in the workload.
*/
{
int size = 0;
Vector index_names = null;
try {
index_names = _analysis.getUselessPhysicalIndexes();
} catch( SQLException e ) {
e.printStackTrace();
return 0;
}
if( index_names == null ) return 0;
for( int i = 0; i < index_names.size(); i++ ) {
int tmp = 0;
try {
tmp = lxtDB.getPhysicalIndexStats((String)index_names.get( i ));
} catch( SQLException e ) {
tmp = 0;
}
if( tmp > 0 ) {
size += tmp;
}
}
return size;
}
public
boolean inProgress()
{
return _in_progress;
}
public
boolean isDone()
{
return _is_done;
}
public
boolean canStopEarly()

```

```

{
return _can_stop_early;
}
public
void cancel()
{
_cancelTuning = true;
}
public void setFeedbackControl( IxtFeedback feedback )
{
_feedbackControl = feedback;
}
public IxtDriver( boolean clustered_option,
boolean keep_existing_option,
long size_constraint,
IxtAnalysis analysis,
Connection connection )
throws SQLException
/* Initialize the driver. Use the options specified from the GUI.
* Get a new master_id to keep track of this analysis.
*/
{
_global = this;
_clustered_option = clustered_option;
_keep_existing_option = keep_existing_option;
_size_constraint = size_constraint;
_in_progress = false;
_is_done = false;
_can_stop_early = false;
_cancelTuning = false;
// Default feedback control - publishes no feedback
// setFeedbackControl method must be used to get feedback
_feedbackControl = new IxtFeedback();
_analysis = analysis;
_stmt = connection.createStatement();
IxtDB.setStatement( _stmt );
//IxtDB.runTruncateStatement();
IxtDB.runStopIndexTuningStatement();
_master_id = IxtDB.getNewMasterID( analysis.getName() );
analysis.setID( _master_id );
IxtDB.debug( IxtDriver.getI18NMessage( IxtDriver.GOT_NEW_MASTER_ID ) + " " + Long.toString( _master_id ) );
}
public
int tune()
throws SQLException
/* The main driver method; loads a workload, and keeps tuning and paring
* down the list of indexes until a solution satisfying the user-specified
* parameters is reached.
*/
{

```

```

//starts a new thread
//the workload must be loaded by this point
lxtInstance instance;
int curPhaseNum = 1;
_in_progress = true;
if( _cancelTuning ) {
    _in_progress = false;
    _is_done = true;
    return 0;
}
showStatus( lxtDriver.getl18NMessage( lxtDriver.GENERATING_INDEXES ) );
lxtDB.runRecommendIndexesStatement( _master_id,
    1,
    _clustered_option,
    _keep_existing_option );
showStatus( lxtDriver.getl18NMessage( lxtDriver.GENERATING_STRUCTURES ) );
instance = new lxtInstance();
instance.discardQueries();
showStatus( lxtDriver.getl18NMessage( lxtDriver.GETTING_PHASE_ONE ) );
_curphase = new lxtPhase( 1, instance, _keep_existing_option );
showStatus( lxtDriver.getl18NMessage( lxtDriver.FOLDING_INDEXES ) );
//this will ensure we do not recommend indexes/pkey/fkeys that already exist
_curphase.augmentIndexesWithPhysical();
_curphase.foldIndexes( new lxtPhase.DuplicateFoldMatcher() );
_curphase.clearNegatives();
_curphase.assignIndexPenalties();
showStatus( lxtDriver.getl18NMessage( lxtDriver.SUBSUMING_INDEXES ) );
_curphase.foldIndexes( new lxtPhase.SubsumingFoldMatcher() );
_curphase.clearNegatives();
_curphase.selectClustered();
_curphase.addReport();
_feedbackControl.setProgressComplete();
_feedbackControl.showResults( curPhaseNum, _curphase );
while( !_cancelTuning && ( _curphase.getTotalSize() - getSizeOfUselessPhysicalIndexes() ) > _size_constraint )
{
    _feedbackControl.setProgressRestart();
    showStatus( lxtDriver.getl18NMessage( lxtDriver.GENERATING_NEW_PHASE ) );
    curPhaseNum++;
    _curphase = _curphase.genNewPhase();
    showStatus( lxtDriver.getl18NMessage( lxtDriver.ELIM_DUP ) );
    _curphase.foldIndexes( new lxtPhase.DuplicateFoldMatcher() );
    showStatus( lxtDriver.getl18NMessage( lxtDriver.ACCOUNTING_FOR_UPDATES ) );
    _curphase.assignIndexPenalties();
    showStatus( lxtDriver.getl18NMessage( lxtDriver.SUBSUMING_INDEXES ) );
    _curphase.foldIndexes( new lxtPhase.SubsumingFoldMatcher() );
    showStatus( lxtDriver.getl18NMessage( lxtDriver.TRIMMING_EXCESS ) );
    _curphase.trimPhase( _size_constraint, PHASE_REDUCTION );
    _curphase.addReport();
    _feedbackControl.setProgressComplete();
    _feedbackControl.showResults( curPhaseNum, _curphase );
}

```

```

}
if( _cancelTuning ) {
    // Commit tuning analysis information
    _stmt.getConnection().commit();
    _in_progress = false;
    _is_done = true;
    return 0;
}
_feedbackControl.setProgressRestart();
showStatus( lxtDriver.getl18NMessage( lxtDriver.GENERATING_SUMMARY ) );
curPhaseNum++;
_curphase = _curphase.genNewPhase();
showStatus( lxtDriver.getl18NMessage( lxtDriver.ACCOUNTING_FOR_UPDATES ) );
_curphase.foldIndexes( new lxtPhase.DuplicateFoldMatcher() );
_curphase.assignIndexPenalties();
showStatus( lxtDriver.getl18NMessage( lxtDriver.WRITING_REPORT ) );
_curphase.addReport();
_feedbackControl.setProgressComplete();
_feedbackControl.showResults( curPhaseNum, _curphase);
lxtDB.runStopIndexTuningStatement();
// Commit tuning analysis information
_stmt.getConnection().commit();
_in_progress = false;
_is_done = true;
return 0;
}
}
// lxtElement.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****
package com.sybase.indexConsultant;
import java.util.*;
public class lxtElement
{
    static final int ASC = 1;
    static final int INSENSITIVE = 0;
    static final int DESC = -1;
    lxtColumn _column;
    int _direction;
    public lxtElement( lxtColumn column, int direction )
    {
        _column = column;
        _direction = direction;
    }
    public
    lxtColumn getColumn()
    {
        return _column;

```

```

    }
    public
    int getDirection()
    {
return _direction;
    }
    public
    void setDirection( int dir )
    {
_direction = dir;
    }
    public
    int getDominantDirection( lxtElement other )
    {
int ret = _direction + other._direction;
if( ret > ASC )
{
    ret = ASC;
}
else if( ret < DESC )
{
    ret = DESC;
}
return ret;
    }
    static class ExactComparator
    implements Comparator
    /* Check whether two index columns are identical */
    {
int _flip;
public ExactComparator( int flip )
{
    _flip = flip;
}
public int compare( Object o1, Object o2 )
{
    lxtElement self = (lxtElement) o1;
    lxtElement other = (lxtElement) o2;
    if( self._column == other._column
        && ( self._direction == INSENSITIVE
            && other._direction == INSENSITIVE )
        || self._direction * other._direction * _flip == ASC ) )
    {
return 0;
    }
    else
    {
return 1;
    }
}
}

```

```

public boolean equals( Object o )
{
    return false;
}

static class RoughComparator
implements Comparator
/* Check whether two index columns are over the same base table and that
 * they do not both have specified, opposite directions */
{
public int compare( Object o1, Object o2 )
{
    lxtElement self = (lxtElement) o1;
    lxtElement other = (lxtElement) o2;
    if( self._column == other._column
        && self._direction * other._direction >= INSENSITIVE )
    {
return 0;
    }
    else
    {
return 1;
    }
}
public boolean equals( Object o )
{
    return false;
}
}

// lxtIndex.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.util.*;
import com.sybase.util.*;
public class lxtIndex
/* This class represents an index, either physical or virtual
 */
{
    static final double UPDATE_COST = 5e07;
    static final int CLUSTERED_UPDATE_PENALTY = 4;
    static final double SIZE_PENALTY_EXPONENT = 0.6;
    long _id;
    lxtTable _table;
    long _size;
    boolean _virtual;
    String _name;

```

```

boolean _is_clustered;
double _cluster_score;
boolean _disabled;
Vector _elements;
Vector _configurations;
double _duipenalty;
boolean _is_key;
public IxtIndex( long id,
    IxtTable table,
    long size,
    boolean virtual,
    String name,
    boolean is_key )
{
_id = id;
_table = table;
_size = size;
_virtual = virtual;
_name = name;
_is_key = is_key;
_disabled = false;
_elements = new Vector();
_configurations = new Vector();
}
public
IxtTable getTable()
{
return _table;
}
public
long getID()
{
return _id;
}
public
boolean isKey()
{
return _is_key;
}
public
void setID( long id )
{
//this should only be used for subsuming indexes
_id = id;
}
public
boolean isClustered()
{
return _is_clustered;
}

```

```

public
    lxtElement getElement( int i )
    {
return (lxtElement) _elements.elementAt(i);
    }
public
    int getNumElements()
    {
return _elements.size();
    }
public
    Iterator getElementIterator()
    {
return _elements.iterator();
    }
public
    void setElements( Vector e )
    {
_elements = e;
    }
public
    void disable()
    {
_disabled = true;
    }
public
    void setIsClustered( boolean val )
    {
_is_clustered = val;
    }
public
    void setSize( long size )
    {
_size = size;
    }
public
    double getClusterScore()
    {
return _cluster_score;
    }
public
    void setClusterScore( double score )
    {
_cluster_score = score;
    }
public
    void addElement( lxtElement elem )
    {
_elements.add( elem );
    }

```

```

    lxtElement getIndexElement( int i )
    {
return (lxtElement) _elements.elementAt( i );
    }
    public
    void addConfiguration( lxtConfiguration config )
    {
_configurations.add( config );
    }
    public
    Vector getConfigurations()
    {
return _configurations;
    }
    public
    void appendConfigurations( Vector configs )
    {
_configurations.addAll( configs );
    }
    public
    void clearConfigurations()
    {
_configurations = new Vector();
    }
    public
    Iterator getConfigurationIterator()
    {
return _configurations.iterator();
    }
    public
    boolean isVirtual()
    {
return _virtual;
    }
    public
    void setIsVirtual( boolean v )
    {
_virtual = v;
    }
    public
    boolean isDisabled()
    {
return _disabled;
    }
    public
    String generateName( long id )
    {
StringBuffer buff = new StringBuffer();
buff.append( "_" );
if( _is_clustered )

```

```

{
    buff.append( "cl" );
}
buff.append( "$" + _table.getName() );
buff.append( "$" + getElement(0).getColumn().getName() );
// id should only be as large as a short integer in C
buff.append( Long.toString( 4294967295L - id ) );
return new String( buff );
}

public
String getName()
{
if( _name == null )
{
    return generateName( _id );
}
else
{
    return _name;
}
}

public
void setName( String name )
{
_name = name;
}

public
long getSize()
{
return _size;
}

public
String genCreateStatement( boolean makephysical,
    String identifier )
{
int i;
StringBuffer buff = new StringBuffer();
buff.append( "create " );
if( !makephysical )
{
    buff.append( " virtual " );
}
if( _is_clustered )
{
    buff.append( " clustered " );
}
buff.append( "index " );
if( _name != null )
{
    buff.append( _name );

```

```

}
else if( identifier != null )
{
    buff.append( identifier );
}
else
{
    buff.append( "Virtual" );
    buff.append( _id );
}
buff.append( " on " );
buff.append( _table.getCreator() + "." + _table.getName() );
buff.append( " ( " );
for( i = 0; i < _elements.size(); i++ )
{
    if( i > 0 )
    {
        buff.append( ", " );
    }
    buff.append( getIndexElement(i).getColumn().getName() );
}
buff.append( ");" );
return new String( buff );
}
public
IxtElement[] GetCommonOrder( IxtIndex other )
{
return null;
}
public
double getDUIPenalty()
{
return _duipenalty;
}
public
double getBenefitSum()
{
double benefit = 0.0;
IxtConfiguration cfg;
Iterator iter = _configurations.iterator();
while( iter.hasNext() )
{
    cfg = (IxtConfiguration) iter.next();
    benefit += cfg.getBenefit();
}
return benefit;
}
public
double getTotalBenefit()
{

```

```

double benefit = getBenefitSum();
benefit -= _duipenalty;
return benefit;
}
/*
 * Compute the relative benefit for a table.
 * The factors involved:
 * 1) Table Size (positive benefit, bigger tables better)
 * 2) # Queries Affected (positive benefit, more queries better)
 * 3) Update DML Statements Affected (negative benefit)
 * 4) Clustered or Non-Clustered (clustered is better)
 * 5) Optimizer Benefit (positive)
 *
 * The idea is, if two indexes are on the same set of queries then
 * we need some way of deciding which one benefits more. This method
 * calculates this on a scale of 1-10 based on the above factors.
 * One factor that was not included because the information was not
 * available was index density - if this becomes available it would
 * be beneficial to add it as a factor below.
 *
 * The way it works:
 * For each factor, its largest value among all of the indexes for
 * this analysis is determined. Then, a line is created to some
 * absolute low (ie. 0 in the case of table size) with a slope and
 * intercept. Then this index's values are used to determine where it
 * fits on the line. If the factor has negative benefits, then the
 * line is given a negative slope with the highest value receiving a
 * ranking of 1 and the lowest value receiving a rank of 10. The opposite
 * would be true for a factor that has postive benefits.
 *
 * After all of these values are determined, the factors are assigned
 * weights relative to one another. Below, table size has a big role
 * so it receives a rather large rating (of 40). Clustered has a smaller
 * role so it receives a ranking of 20. These numbers can be adjusted
 * as required to make the outcome make more sense if it is determined
 * that they do not produce appropriate output.
 *
 * After the weights are applied, they are summed up and divided by the
 * total of all the weights - thus giving a final result between 1 and 10.
 */
public
double getRelativeBenefit()
{
    lxtPhase phase = lxtDriver.getCurrentPhase();
    double updateCostRating = 0.0;
    if( phase.getMaxDUIPenalty() > 0.0 )
    {
        updateCostRating = -20.0 * _duipenalty / (phase.getMaxDUIPenalty() );
    }
    double queriesAffectedRating = 0.0;

```

```

if( phase.getMaxNumConfigs() > 0 )
{
    queriesAffectedRating = 20.0 * _configurations.size() / (phase.getMaxNumConfigs() );
}
double tableSizeRating = 40.0 * _table.getSize() / (phase.getInstance().getMaxTableSize() );
double benefitRating = 0.0;
if( phase.getMaxBenefitSum() > 0 )
{
    benefitRating = 35.0 * getBenefitSum() / (phase.getMaxBenefitSum() );
}
double clusteredRating = ( _is_clustered ? 20.0 : 1.0 );
double ret = tableSizeRating + clusteredRating + benefitRating;
ret += updateCostRating + queriesAffectedRating;
// divide by sum of weights, then multiply by 10 to get a rank
// between 1 and 10 (10 is best)
ret = ret / 13.5;
if( ret < 1.0 ) ret = 1.0;
if( ret > 10.0 ) ret = 10.0;
if( ret == Double.NaN )
{
    ret = 1.0;
    Dbg.wassert( false );
}
return ret;
}

public
boolean isOnTable( lxtTable tab )
{
return _table == tab;
}

public
boolean containsColumn( lxtColumn col )
{
lxtElement element;
Iterator iter;
if( _table != col.getTable() )
{
    return false;
}
iter = _elements.iterator();
while( iter.hasNext() )
{
    element = (lxtElement) iter.next();
    if( element.getColumn() == col )
    {
return true;
    }
}
return false;
}

```

```

public
void addDUIPenalty( double numRows )
{
//TODO - what if lots of rows are on the same pages?
_duipenalty += numRows * UPDATE_COST * Math.log( _size );
if( _is_clustered )
{
    _duipenalty *= CLUSTERED_UPDATE_PENALTY;
}
}

static class ExactComparator
implements Comparator
{
public int compare( Object o1, Object o2 )
{
    int i;
    IxtIndex self = (IxtIndex) o1;
    IxtIndex other = (IxtIndex) o2;
    if( self._table.getID() != other._table.getID()
|| self._elements.size() != other._elements.size() )
    {
return 1;
    }
    for( i = 0; i < self._elements.size(); i++ )
    {
if( self.getIndexElement(i).getColumn()
    != other.getIndexElement(i).getColumn() )
    {
return 1;
    }
    }
    return 0;
}

public boolean equals( Object o )
{
    return false;
}

}

public Vector getCommonOrder( IxtIndex other )
{
int INSENSITIVE = 0;
int ASC = 1;
int DESC = -1;
Vector aInsensitive = new Vector();
Vector bInsensitive = new Vector();
Vector aFixed = new Vector();
Vector bFixed = new Vector();
boolean firstSet = false;
int flip = INSENSITIVE;
Vector finalOrder = new Vector();

```

```

Iterator aelemIter;
Iterator belemIter;
lxtElement aelem;
lxtElement belem;
int dir = INSENSITIVE;
aelemIter = getElementIterator();
while( aelemIter.hasNext() )
{
    aelem = (lxtElement)( aelemIter.next() );
    if( !firstSet && aelem.getDirection() == INSENSITIVE )
    {
        aInsensitive.add( aelem );
    }
    else
    {
        firstSet = true;
        aFixed.add(aelem);
    }
}
firstSet = false;
belemIter = other.getElementIterator();
while( belemIter.hasNext() )
{
    belem = (lxtElement) belemIter.next();
    if( !firstSet && belem.getDirection() == INSENSITIVE )
    {
        bInsensitive.add( belem );
    }
    else
    {
        firstSet = true;
        bFixed.add( belem );
    }
}
if( aInsensitive.size() < bInsensitive.size() )
{
    // swap a and b, so we will know that len(alns) >= len(blns)
    Vector temp = aInsensitive;
    aInsensitive = bInsensitive;
    bInsensitive = temp;
    temp = aFixed;
    aFixed = bFixed;
    bFixed = temp;
}
belemIter = bInsensitive.iterator();
while( belemIter.hasNext() )
{
    belem = (lxtElement) belemIter.next();
    aelemIter = aInsensitive.iterator();
    while( aelemIter.hasNext() )

```

```

    {
    aelem = (IxtElement) aelemIter.next();
    if( new IxtElement.ExactComparator( INSENSITIVE ).compare( belem, aelem ) == 0 )
    {
        finalOrder.add( new IxtElement( aelem.getColumn(), INSENSITIVE ) );
        aelemIter.remove();
        break;
    }
    //these orders have different prefixes an so cannot match
    return null;
    }
}
// now we are left with no unmatched bInsensitives
belemIter = bFixed.iterator();
while( belemIter.hasNext() )
{
    belem = (IxtElement) belemIter.next();
    if( aInsensitive.size() > 0 )
    {
    aelemIter = aInsensitive.iterator();
    while( aelemIter.hasNext() )
    {
        aelem = (IxtElement) aelemIter.next();
        if( new IxtElement.RoughComparator().compare( belem, aelem ) == 0 )
        {
            dir = aelem.getDominantDirection( belem );
            finalOrder.add( new IxtElement( belem.getColumn(), dir ) );
            aelemIter.remove();
            break;
        }
        // no match is possible
        return null;
    }
    }
    else if( aFixed.size() > 0 )
    {
    aelem = (IxtElement) aFixed.elementAt(0);
    // may be overridden below
    dir = belem.getDirection();
    if( flip == INSENSITIVE && new IxtElement.ExactComparator( ASC ).compare( belem, aelem ) == 0 )
    {
        flip = ASC;
    }
    else if( flip == INSENSITIVE && new IxtElement.ExactComparator( DESC ).compare( belem, aelem ) == 0 )
    {
        flip = DESC;
    }
    else if( new IxtElement.ExactComparator( flip ).compare( belem, aelem ) == 0 )
    {
        //do nothing

```

```

    }
    else if( new lxtElement.RoughComparator().compare( belem, aelem ) == 0 )
    {
        dir = belem.getDominantDirection( aelem );
    }
    else
    {
        //can't match
        return null;
    }
    }
    else
    {
        finalOrder.add( new lxtElement( belem.getColumn(), dir ) );
    }
}
if( aFixed.size() > 0 )
{
    aelemIter = aFixed.iterator();
    while( aelemIter.hasNext() )
    {
        aelem = (lxtElement) aelemIter.next();
        finalOrder.add( new lxtElement( aelem.getColumn(), aelem.getDirection() ) );
    }
}
if( flip == DESC )
{
    aelemIter = finalOrder.iterator();
    while( aelemIter.hasNext() )
    {
        aelem = (lxtElement) aelemIter.next();
        aelem.setDirection( aelem.getDirection() * DESC );
    }
}
return finalOrder;
}
static class SizeComparator
implements Comparator
{
public int compare( Object o1, Object o2 )
{
    lxtIndex self = (lxtIndex) o1;
    lxtIndex other = (lxtIndex) o2;
    return new Long( self._size ).compareTo( new Long( other._size ) );
}
public boolean equals( Object o )
{
    return false;
}
}

```

```

static class TableComparator
implements Comparator
{
public int compare( Object o1, Object o2 )
{
    lxtIndex self = (lxtIndex) o1;
    lxtIndex other = (lxtIndex) o2;
    if( self._table == other._table )
    {
return 0;
    }
    else
    {
return 1;
    }
}
public boolean equals( Object o )
{
    return false;
}
}

static class TotalBenefitComparator
implements Comparator
{
public int compare( Object o1, Object o2 )
{
    lxtIndex self = (lxtIndex) o1;
    lxtIndex other = (lxtIndex) o2;
    return Double.compare( self.getTotalBenefit(), other.getTotalBenefit() );
}
public boolean equals( Object o )
{
    return false;
}
}

static class RelativeBenefitComparator
implements Comparator
{
public int compare( Object o1, Object o2 )
{
    lxtIndex self = (lxtIndex) o1;
    lxtIndex other = (lxtIndex) o2;
    return Double.compare( self.getRelativeBenefit(), other.getRelativeBenefit() );
}
public boolean equals( Object o )
{
    return false;
}
}
}

```

```

// IxtInstance.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.util.*;
import java.sql.*;
public class IxtInstance
{
    static final double UNUSABLE_COST = 1e32;
    static final double MINIMUM_RELEVANCE_RATIO = 0.01;
    HashMap _tables;
    HashMap _queries;
    Vector _physicalindexes;
    Vector _discardablequeries;
    int _max_table_size;
    public IxtInstance()
    throws SQLException
    {
        _tables = new HashMap();
        _queries = new HashMap();
        _physicalindexes = new Vector();
        _discardablequeries = new Vector();
        _max_table_size = 0;
        loadTables();
        loadColumns();
        loadIndexes();
        loadQueries();
        selectDiscardableQueries();
    }
    public
    IxtQuery findQuery( long id )
    {
        return (IxtQuery) _queries.get( new Long( id ) );
    }
    public
    IxtTable findTable( long id )
    {
        return (IxtTable) _tables.get( new Long( id ) );
    }
    public
    HashMap getTables()
    {
        return _tables;
    }
    public
    int getMaxTableSize()
    {
        return _max_table_size;
    }
}

```

```

    }
    public
    Iterator getQueryIterator()
    {
return _queries.values().iterator();
    }
    public
    Iterator getDiscardableIterator()
    {
return _discardablequeries.iterator();
    }
    public
    Iterator getPhysicalIndexIterator()
    {
return _physicalindexes.iterator();
    }
    public
    Vector getPhysicalIndexes()
    {
return _physicalindexes;
    }
    void loadTables()
    throws SQLException
    /* Get all tables from the database catalog and store lxtTable objects for
     * them.
     */
    {
lxtTable tab;
ResultSet res = lxtDB.runTableQuery();
while( res.next() )
{
    tab = new lxtTable( res.getLong(1),
        res.getString(2),
        res.getString(3),
        res.getInt(4),
        res.getInt(5) );
    _tables.put( new Long(res.getLong(1)), tab );
    if( res.getInt(5) > _max_table_size )
    {
_max_table_size = res.getInt(5);
    }
}
res.close();
    }
    void loadColumns()
    throws SQLException
    /* Get all columns from the database catalog, create lxtColumns objects for
     * them, and assign them to the appropriate lxtTable objects.
     */
    {

```

```

lxtColumn col;
ResultSet res;
lxtTable tab;
lxtIndex pkix;
Iterator iter = _tables.values().iterator();
while( iter.hasNext() )
{
    tab = (lxtTable)iter.next();
    pkix = new lxtIndex( 0, tab, 0, false, "PKEY", true );
    res = lxtDB.runColumnsQuery( tab.getID() );
    while( res.next() )
    {
        col = new lxtColumn( res.getLong(1), res.getString(2), tab );
        tab.addColumn( col );
        if( res.getInt(3) == 1 )
        {
            pkix.addElement( new lxtElement( col, lxtElement.ASC ) );
        }
        if( pkix.getNumElements() > 0 ) {
            tab.addPhysicalIndex( pkix );
            _physicalindexes.add( pkix );
        }
        res.close();
    }
}

void loadIndexes()
throws SQLException
/* Get all indexes (both key and secondary indexes), create index objects
 * for them, and assign them to the appropriate table objects.
 */
{
    lxtTable tab;
    lxtIndex idx;
    Iterator iter;
    ResultSet res = lxtDB.runPhysicalIndexesQuery();
    while( res.next() )
    {
        tab = findTable( res.getLong(2) );
        idx = new lxtIndex( res.getLong(1), tab, 0, false, res.getString(3), res.getInt(4) == 1 );
        tab.addPhysicalIndex( idx );
        _physicalindexes.add( idx );
    }
    res.close();
    iter = _physicalindexes.iterator();
    while( iter.hasNext() )
    {
        idx = (lxtIndex) iter.next();
        if( idx.isKey() )
        {

```

```

res = lxtDB.runKeyIndexColumnsQuery( idx.getTable().getID(), idx.getID() );
    }
    else
    {
res = lxtDB.runPhysicalIndexColumnsQuery( idx.getTable().getID(),
    idx.getID() );
    }
    while( res.next() )
    {
idx.addElement( new lxtElement(
    idx.getTable().findColumn( res.getLong(1) ),
    res.getInt(2) )
);
    }
    res.close();
}
iter = _tables.values().iterator();
while( iter.hasNext() )
{
    tab = (lxtTable) iter.next();
    if( tab.getClusterID() != 0 )
    {
idx = tab.findPhysicalIndex( tab.getClusterID() );
if( idx != null )
{
    idx.setIsClustered( true );
}
    }
}
}
void loadQueries()
throws SQLException
/* Get all queries in the workload and load them. Load all columns and
 * tables affected by each query (from ix_consultant_affected_columns) and
 * assign queries affecting a table or column to that table or column object
 */
{
lxtQuery query;
lxtTable tab;
Iterator iter;
long coln;
ResultSet res = lxtDB.runQueriesQuery();
while( res.next() )
{
    query = new lxtQuery( res.getLong(1),
        res.getString(2).charAt(0),
        res.getDouble(3),
        res.getDouble(4),
        "" );
    _queries.put( new Long(query.getID()), query );
}

```

```

}
res.close();
res = lxtDB.runWorkloadQuery();
while( res.next() )
{
    query = findQuery( res.getLong(1) );
    //TODO: this should probably be removed, but for now it
    //prevents us from crashing if there was a problem optimizing
    //a given statement
    if( query == null ) { continue; }
    query.addWorkloadItem( res.getLong(2), res.getLong(3) );
}
res.close();
iter = _queries.values().iterator();
while( iter.hasNext() )
{
    query = (lxtQuery) iter.next();
    res = lxtDB.runAffectedColumnsQuery( query.getID() );
    while( res.next() )
    {
        tab = findTable( res.getLong(1) );
        coln = res.getLong(2);
        if( coln == 0 )
        {
            query.addAffectedTable( tab );
        }
        else
        {
            query.addAffectedColumn( tab.findColumn( coln ) );
        }
    }
    res.close();
}
}

void selectDiscardableQueries()
/* Add queries that are below a certain cost threshold to a list of queries
 * to be discarded
 */
{
double maxcost = 0;
lxtQuery query;
Iterator iter;
iter = _queries.values().iterator();
while( iter.hasNext() )
{
    query = (lxtQuery) iter.next();
    if( query.getVanillaCost() < UNUSABLE_COST )
    {
        maxcost = Math.max( query.getVanillaCost() * query.getCount(),
            maxcost );
    }
}
}

```

```

    }
    else
    {
        _discardablequeries.add( query );
    }
}

}

public
double getTotalVanillaCost()
{
double ret = 0.0;
Iterator iter = _queries.values().iterator();
IxtQuery query;
while( iter.hasNext() )
{
    query = (IxtQuery)iter.next();
    ret += query.getVanillaCost();
}
return ret;
}

public
void discardQueries()
throws SQLException
/* For each query in the "discarded" list, actually discard it from the
 * workload
 */
{
if( _discardablequeries.size() > 0 )
{
    IxtDB.runDiscardWorkloadItemsStatement( getDiscardableIterator() );
}
}
}

// IxtPhase.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.util.*;
import java.sql.*;
import com.sybase.util.*;
public class IxtPhase
{
    int    _id;
    IxtInstance    _instance;
    Vector    _configurations;
    Vector    _indexes;
    int    _max_num_configs;
    double    _max_benefit_sum;

```

```

double    _max_dui_penalty;
/**
 * Keep Existing requires special handling. After fixing the "Unused"
 * information in the engine by adding all physical indexes to the
 * ix_consultant_index table, we required special handling of those
 * recommended that are physical indexes. Previously, if keep existing
 * was off then the treatment of a physical index was identical to that of
 * a virtual index. So, when disabling or discarded indexes we must
 * change it to only do so when the index is virtual OR keep existing
 * is off.
 */
boolean    _keep_existing;
public IxtPhase( int id, IxtInstance instance, boolean keep_existing )
throws SQLException
{
    Iterator cfgiter;
    Iterator indexiter;
    IxtIndex index;
    IxtConfiguration config;
    _instance = instance;
    _id = id;
    _configurations = new Vector();
    _indexes = new Vector();
    _keep_existing = keep_existing;
    getConfigurations();
    getIndexes();
    cfgiter = _configurations.iterator();
    while( cfgiter.hasNext() )
    {
        config = (IxtConfiguration) cfgiter.next();
        indexiter = config.getIndexIterator();
        while( indexiter.hasNext() )
        {
            index = (IxtIndex) indexiter.next();
            _indexes.add( index );
        }
    }
    _max_num_configs = 0;
    _max_benefit_sum = 0;
    _max_dui_penalty = 0;
}

void getConfigurations()
throws SQLException
/* Get all configurations for the current phase (that is, bindings between
 * queries from the workload and indexes that were picked for them during
 * this iteration.
 * showSubStatus methods are used to give feedback to user about where in
 * the process we currently are.
 */
{

```

```

long iid;
lxtTable tab;
long npages;
double score1;
boolean is_clstrd;
double cluster_score;
boolean is_virt;
lxtConfiguration config;
lxtQuery query;
lxtIndex idx = null;
ResultSet res;
Vector tempindexes;
Iterator iter = _instance.getQueryIterator();
int i = 0;
lxtDriver.showSubstatus( lxtDriver.getl18NMessage( lxtDriver.GETTING_CONFIGURATIONS ) );
while( iter.hasNext() )
{
    query = (lxtQuery) iter.next();
    res = lxtDB.runConfigurationsQuery( _id, query.getID() );
    score1 = query.getVanillaCost();
    tempindexes = new Vector();
    i++;
    lxtDriver.showSubstatus( lxtDriver.getl18NMessage( lxtDriver.GETTING_CONFIGURATION ) + " " +
Integer.toString( i ) );
    while( res.next() )
    {
        iid = res.getLong(1);
        tab = _instance.findTable( res.getLong(2) );
        npages = res.getLong(3);
        score1 = res.getDouble(4);
        is_clstrd = (res.getInt(5) != 0);
        cluster_score = res.getDouble(6);
        is_virt = (res.getInt(7) != 0);
        if( !is_virt )
        {
            idx = tab.findPhysicalIndex( iid );
            idx.setSize( npages );
            // Clear configurations for this phase
            if( idx.getConfigurations().size() > 0 ) {
            idx.clearConfigurations();
            }
        }
        else
        {
            idx = new lxtIndex( iid, tab, npages, is_virt, null, false );
        }
        idx.setClusterScore( cluster_score );
        idx.setIsClustered( is_clstrd );
        tempindexes.add( idx );
    }
}

```

```

        res.close();
        if( score1 >= query.getVanillaCost() )
        {
//DJFTODO - we must drop this query
score1 = query.getVanillaCost();
        }
        config = new lxtConfiguration( query, score1, tempindexes );
        _configurations.add( config );
        query.setConfig( _id, config );
    }
}

public
void augmentIndexesWithPhysical()
{
_indexes.addAll( _instance.getPhysicalIndexes() );
}
void getIndexes()
throws SQLException
/* Get all indexes recommended in this phase
*/
{
Iterator indexiter;
Iterator cfgiter = _configurations.iterator();
lxtConfiguration config;
lxtIndex index;
ResultSet res;
int i = 0;
while( cfgiter.hasNext() )
{
    config = (lxtConfiguration) cfgiter.next();
    indexiter = config.getIndexIterator();
    lxtDriver.showSubstatus( lxtDriver.getl18NMessage( lxtDriver.GETTING_VIRT_INDEXES ) );
    while( indexiter.hasNext() )
    {
index = (lxtIndex) indexiter.next();
i++;
lxtDriver.showSubstatus( lxtDriver.getl18NMessage( lxtDriver.GETTING_VIRT_INDEX ) + " " + Integer.toString( i ) );
res = lxtDB.runIndexColumnsQuery( _id,
        index.getTable().getID(),
        index.getID() );
while( res.next() )
{
    index.addElement( new lxtElement(
index.getTable().findColumn( res.getLong(1) ),
        res.getInt(2) )
        );
    }
res.close();
index.addConfiguration( config );
}
}

```

```

}
}
public
void assignIndexPenalties()
/* For each index over tables or columns that have insert/update/delete
 * operations in the workload, assign a penalty to the index
 */
{
lxtQuery query;
lxtTable tab;
lxtColumn col;
lxtIndex index;
Iterator query_iter;
Iterator tab_iter;
Iterator idx_iter;
Iterator col_iter;
int i = 0;
query_iter = _instance.getQueryIterator();
while( query_iter.hasNext() )
{
    query = (lxtQuery) query_iter.next();
    i++;
    lxtDriver.showSubstatus( lxtDriver.getl18NMessage( lxtDriver.ACCOUNTING_FOR_UPDATE ) + " " +
Integer.toString( i ) );
    tab_iter = query.getAffectedTablesIterator();
    while( tab_iter.hasNext() )
    {
tab = (lxtTable) tab_iter.next();
idx_iter = _indexes.iterator();
while( idx_iter.hasNext() )
{
    index = (lxtIndex)idx_iter.next();
    if( index.isOnTable( tab ) )
    {
index.addDUIPenalty( query.getCount()
    * query.getNumRowsAffected() );
    }
}
}
col_iter = query.getAffectedColumnsIterator();
while( col_iter.hasNext() )
{
col = (lxtColumn) col_iter.next();
idx_iter = _indexes.iterator();
while( idx_iter.hasNext() )
{
    index = (lxtIndex) idx_iter.next();
    if( index.containsColumn( col ) )
    {
index.addDUIPenalty( query.getCount()

```

```

        * query.getNumRowsAffected() );
    }
}
}
}
public
    IxtPhase genNewPhase()
        throws SQLException
    /* Ask for and then build the next phase of tuning
    */
    {
IxtDB.runStopIndexTuningStatement();
generateIndexes();
IxtDB.runRecommendIndexesStatement( IxtDriver.getMasterID(),
    _id + 1,
    IxtDriver.getClusteredOption(),
    IxtDriver.getKeepExistingOption() );
return new IxtPhase( _id + 1, _instance, _keep_existing );
    }
    public
        double getTotalPhaseCost()
    /* Get the total cost for the current phase (using all recommended indexes
    * belonging to this phase)
    */
    {
IxtQuery query;
IxtConfiguration config;
double ret = 0.0;
Iterator iter = _instance.getQueryIterator();
while( iter.hasNext() )
{
    query = (IxtQuery) iter.next();
    {
config = query.getConfigurationByPhase( _id );
if( config != null )
{
    ret += config.getWorkingCost();
}
else
{
    ret += query.getVanillaCost();
}
}
}
return ret;
    }
    public
        void generateIndexes()
            throws SQLException

```

```

/* Generate the virtual indexes required for the next phase (since the
 * optimizer only generates virtual indexes on the first phase - all
 * subsequent phases must supply their own list of virtual indexes back to
 * the optimizer). Also, tell the optimizer to ignore physical indexes that
 * have been disabled.
 */
{
Iterator iter;
IxtIndex index;
String str;
iter = _indexes.iterator();
while( iter.hasNext() )
{
    index = (IxtIndex) iter.next();
    if( index.isVirtual() )
    {
        str = index.genCreateStatement( false, null );
        IxtDB.getStatement().execute( str );
    }
}
iter = _instance.getPhysicalIndexIterator();
while( iter.hasNext() )
{
    index = (IxtIndex) iter.next();
    if( index.isDisabled()
        && !index.isKey()
        && (index.isVirtual() || !_keep_existing) )
    {
        IxtDB.runDisableIndexStatement( index.getName(),
            index.getTable().getName(),
            index.getTable().getCreator() );
    }
}
}

public
long getTotalSize()
{
    long total = 0;
    Iterator iter = _indexes.iterator();
    IxtIndex index;
    while( iter.hasNext() )
    {
        index = (IxtIndex) iter.next();
        total += index.getSize();
    }
    return total;
}

public
long getTotalRecommendedSize()
{

```

```

long total = 0;
Iterator iter = _indexes.iterator();
IdxIndex index;
while( iter.hasNext() )
{
    index = (IdxIndex) iter.next();
    // Special handling of physical indexes (don't include their
    // size if keep existing is on)
    if( index.isVirtual() || !_keep_existing ) {
total += index.getSize();
    }
}
return total;
}

public
void trimFromFront( double size )
/* Keep removing indexes from the set of indexes until we have trimmed a
 * number of index pages greater than or equal to the specified size.
 */
{
IdxIndex index;
long trimmed_size = 0;
Iterator iter = _indexes.iterator();
while( iter.hasNext() && trimmed_size < size )
{
    index = (IdxIndex) iter.next();
    // Another special treatment of physical indexes when
    // keep existing is on
    if( index.isVirtual() || !_keep_existing ) {
trimmed_size += index.getSize();
index.disable();
iter.remove();
    }
}
}

public
void selectClustered()
/* Mark the index having the greatest cluster score for each table
 * as the clustered index for that table
 */
{
IdxIndex index;
IdxIndex cur_best;
HashMap best_clust = new HashMap();
Iterator iter = _indexes.iterator();
while( iter.hasNext() )
{
    index = (IdxIndex) iter.next();
    if( best_clust.containsKey( index.getTable() ) )
    {

```

```

cur_best = (IxtIndex) best_clust.get( index.getTable() );
if( index.getClusterScore() > cur_best.getClusterScore() )
{
    best_clust.put( index.getTable(), index );
    cur_best.setIsClustered( false );
}
else
{
    index.setIsClustered( false );
}
}
else
{
    best_clust.put( index.getTable(), index );
}
}
}

public
double getImprovementSum()
/* Add the benefit experienced by each query due to recommended virtual indexes
 * for all configurations (queries using virtual indexes) in the phase.
 */
{
IxtConfiguration config;
double ret = 0.0;
Iterator iter = _configurations.iterator();
while( iter.hasNext() )
{
    config = (IxtConfiguration) iter.next();
    ret += config.getBenefit();
}
return ret;
}

public
String makeSQL()
/* Generate the SQL text needed to implement the changes tested in this
 * phase (ie. create recommended indexes and drop unused indexes)
 */
{
StringBuffer buff = new StringBuffer();
long id = 0;
IxtIndex index;
Iterator iter = _indexes.iterator();
while( iter.hasNext() )
{
    index = (IxtIndex) iter.next();
    id += 1;
    buff.append( index.genCreateStatement( true,
        index.generateName(id) ) );
}
}

```

```

iter = _instance.getPhysicalIndexIterator();
while( iter.hasNext() )
{
    index = (IxtIndex) iter.next();
    if( index.isDisabled() && (index.isVirtual() || !_keep_existing) )
    {
        buff.append( "drop index "
            + index.getTable().getCreator()
            + "." + index.getTable().getName()
            + "." + index.getName() );
    }
}
return new String(buff);
}
public
void clearNegatives()
/* Disable any indexes having a negative benefit (which may happen due to
 * heuristic optimizations in the query optimizer).
 */
{
    IxtIndex index;
    Iterator iter = _indexes.iterator();
    while( iter.hasNext() )
    {
        index = (IxtIndex) iter.next();
        if( index.isKey() )
        {
            iter.remove();
        }
        else if( index.getTotalBenefit() <= 0.0 )
        {
            if( index.isVirtual() || !_keep_existing ) {
                index.disable();
                iter.remove();
            }
        }
    }
}
}
public
void trimPhase( long size_constraint, double phase_reduction )
/* Determine how many pages to trim from this phase, sort the indexes
 * according to benefit, and remove the bottom portion (according to
 * phase_reduction parameter).
 */
{
    computeLimits();
    long total_size = getTotalSize();
    double size_to_trim = total_size * phase_reduction;
    if( ( total_size - size_to_trim ) < size_constraint )
    {

```

```

        size_to_trim = Math.max( total_size - size_constraint, 0 );
    }
    clearNegatives();
    if( ( size_to_trim ) > 0 )
    {
        Collections.sort( _indexes, new IxtIndex.RelativeBenefitComparator() );
        trimFromFront( size_to_trim );
    }
}

public
void computeLimits()
{
    IxtIndex cur;
    ListIterator iter = _indexes.listIterator();
    _max_num_configs = 0;
    _max_benefit_sum = 0;
    _max_dui_penalty = 0;
    while( iter.hasNext() )
    {
        cur = (IxtIndex) iter.next();
        if( cur.getBenefitSum() > _max_benefit_sum )
        {
            _max_benefit_sum = cur.getBenefitSum();
        }
        if( cur.getDUIPenalty() > _max_dui_penalty )
        {
            _max_dui_penalty = cur.getDUIPenalty();
        }
        if( cur.getConfigurations().size() > _max_num_configs )
        {
            _max_num_configs = cur.getConfigurations().size();
        }
    }
}

// The following three methods may only be called after computeLimits()
public
double getMaxBenefitSum()
{
    return _max_benefit_sum;
}

public
int getMaxNumConfigs()
{
    return _max_num_configs;
}

public
double getMaxDUIPenalty()
{
    return _max_dui_penalty;
}

```

```

public
  lxtInstance getInstance()
  {
return _instance;
  }
public
  void foldIndexes( FoldMatcher matcher )
  /* Using the specified matcher, combine indexes that are "similar" (where
   * the definition of similar depends on the matcher used) into a single
   * index structure, assigning all configurations, queries, costs, benefits,
   * etc. of both onto the new single index.
   */
  {
lxtIndex  cur;
lxtIndex  other;
lxtConfiguration  config;
Iterator  vec_iter;
ListIterator  cur_iter;
ListIterator  other_iter;
Iterator  cfg_iter;
boolean remove_flag = false;
HashMap table_group = new HashMap();
lxtTable tab;
Vector  cur_vec;
/* Assumption: Indexes will only fold into other indexes
 * on the same table */
vec_iter = _instance.getTables().values().iterator();
while( vec_iter.hasNext() )
{
  tab = (lxtTable) vec_iter.next();
  table_group.put( new Long( tab.getID() ), new Vector() );
}
cur_iter = _indexes.listIterator();
while( cur_iter.hasNext() )
{
  cur = (lxtIndex) cur_iter.next();
  cur_vec = (Vector) table_group.get(
new Long( cur.getTable().getID() ) );
  cur_vec.add( cur );
}
vec_iter = table_group.values().iterator();
while( vec_iter.hasNext() )
{
  cur_vec = (Vector) vec_iter.next();
  cur_iter = cur_vec.listIterator();
  while( cur_iter.hasNext() )
  {
cur = (lxtIndex) cur_iter.next();
if( cur.isDisabled() )
{

```

```

        continue;
    }
    other_iter = cur_vec.listIterator( cur_iter.nextIndex() );
    while( other_iter.hasNext() )
    {
        other = (IxtIndex) other_iter.next();
        if( other.isDisabled() )
        {
            continue;
        }
        if( cur.getID() == other.getID()
            && cur.getTable().getID() == other.getTable().getID() )
        {
            remove_flag = true;
        }
        else if( matcher.canFold( cur, other ) )
        {
            cfg_iter = other.getConfigurationIterator();
            while( cfg_iter.hasNext() )
            {
                config = (IxtConfiguration) cfg_iter.next();
                config.removeIndex( other );
                config.addIndex( cur );
            }
            cur.appendConfigurations( other.getConfigurations() );
            remove_flag = true;
            if( cur.getName() == null && other.getName() != null )
            {
                cur.setName( other.getName() );
            }
            if( remove_flag )
            {
                if( matcher.isCommutative() && !other.isVirtual() )
                {
                    cur.disable();
                    // we do this to make sure that a virtual index
                    // will not override a physical index
                }
                else if( other.isVirtual() )
                {
                    other.disable();
                    cur.setClusterScore(
                        Math.max( cur.getClusterScore(),
                            other.getClusterScore() )
                    );
                    cur.setIsClustered( cur.isClustered()
                        || other.isClustered() );
                }
            }
            else

```

```

{
    Dbg.wassert( false );
    other.disable();
}
remove_flag = false;
}
}
}
cur_iter = _indexes.listIterator();
while( cur_iter.hasNext() )
{
    cur = (IxtIndex) cur_iter.next();
    if( cur.isDisabled() )
    {
        cur_iter.remove();
    }
}
}
public
void addReport()
throws SQLException
{
StringBuffer buff;
IxtIndex index;
IxtElement elem;
//IxtConfiguration config;
Iterator elem_iter;
Iterator cfg_iter;
Iterator idx_iter = _indexes.iterator();
computeLimits();
// Go through indexes updating benefit/penalty information
while( idx_iter.hasNext() )
{
    buff = new StringBuffer(100);
    index = (IxtIndex) idx_iter.next();
    // Don't bother with physical index stats if _keep_existing
    // is true.
    if( !index.isVirtual() && _keep_existing ) continue;
    elem_iter = index.getElementIterator();
    while( elem_iter.hasNext() )
    {
        elem = (IxtElement) elem_iter.next();
        buff.append( elem.getColumn().getName() );
        if( elem_iter.hasNext() )
        {
            buff.append( ", " );
        }
    }
    IxtDB.runUpdateIndexTable( _id,

```

```

        index.getTable().getID(),
        index.getID(),
        index.isClustered(),
        index.getTotalBenefit(),
        index.getDUIPenalty() );
    }
    // Iterate through indexes updating relative benefit information
    // now that all other information is current
    idx_iter = _indexes.iterator();
    while( idx_iter.hasNext() )
    {
        index = (IxtIndex) idx_iter.next();
        // Don't bother with physical index stats if _keep_existing
        // is true.
        if( !index.isVirtual() && _keep_existing || index.isKey() ) continue;
        IxtDB.runRelativeBenefitUpdateIndexTable( _id,
            index.getTable().getID(),
            index.getID(),
            index.getRelativeBenefit() );
    }
    cfg_iter = _configurations.iterator();
    /*
    while( cfg_iter.hasNext() )
    {
        config = (IxtConfiguration) cfg_iter.next();
        idx_iter = config.getIndexIterator();
        buff = new StringBuffer(100);
        while( idx_iter.hasNext() )
        {
            index = (IxtIndex) idx_iter.next();
            buff.append( index.getName() );
            if( idx_iter.hasNext() )
            {
                buff.append( ", " );
            }
        }
        IxtDB.runUpdateQueryTable( _id,
            config.getQuery().getID(),
            config.getQuery().getType(),
            config.getQuery().getCount(),
            config.getQuery().getText(),
            config.getQuery().getVanillaCost(),
            config.getRealCost(),
            new String(buff) );
    }
    */
    }
    static interface FoldMatcher
    {
        public boolean canFold( IxtIndex cur, IxtIndex other );
    }

```

```

// commutative matcher means that if two indexes match, it doesn't
// matter which one we pick
public boolean isCommutative();
}
static class DuplicateFoldMatcher
implements FoldMatcher
/* This matcher reports indexes as foldable only if they match exactly
*/
{
lxtlIndex.ExactComparator _comparator;
public DuplicateFoldMatcher()
{
_comparator = new lxtlIndex.ExactComparator();
}
public boolean canFold( lxtlIndex cur, lxtlIndex other )
{
return cur != other
&& 0 == _comparator.compare( cur, other );
}
public boolean isCommutative()
{
return true;
}
}
static class SubsumingFoldMatcher
implements FoldMatcher
/* This matcher reports indexes as subsumable only if the other index has an
* ordering that can be subsumed into the current ordering
*/
{
public boolean canFold( lxtlIndex cur, lxtlIndex other )
{
Vector order = cur.getCommonOrder( other );
if( order == null )
{
return false;
}
else
{
cur.setElements( order );
other.setElements( order );
return true;
}
}
public boolean isCommutative()
{
return false;
}
}
}

```

```

// IxtQuery.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.util.*;
public class IxtQuery
{
    static public final char INSERT = 'I';
    static public final char DELETE = 'D';
    static public final char UPDATE = 'U';
    static public final char SELECT = 'S';
    long _id;
    char _type;
    double _vanilla;
    long _count;
    HashMap _configs_by_phase;
    Vector _workloadqueries;
    double _numrowsaffected;
    boolean _discarded;
    Vector _affected_columns;
    String _text;
    Vector _affected_tables;
    public IxtQuery( long id,
        char type,
        double vanilla,
        double numrowsaffected,
        String text )
    {
        _id = id;
        _type = type;
        _vanilla = vanilla;
        _numrowsaffected = numrowsaffected;
        _count = 0;
        _text = text;
        _affected_columns = new Vector();
        _affected_tables = new Vector();
        _configs_by_phase = new HashMap();
        _workloadqueries = new Vector();
    }
    public
    long getID()
    {
        return _id;
    }
    public
    String getText()
    {
        return _text;
    }
}

```

```

    }
    public
    char getType()
    {
return _type;
    }
    public
    void setConfig( int phase, lxtConfiguration config )
    {
_configs_by_phase.put( new Integer( phase ), config );
    }
    public
    lxtConfiguration getConfigurationByPhase( int phase )
    {
return (lxtConfiguration)_configs_by_phase.get( new Integer( phase ) );
    }
    public
    void addWorkloadItem( long workload_id, long count )
    {
_workloadqueries.add( new Long( workload_id ) );
_count += count;
    }
    public
    void addAffectedTable( lxtTable tab )
    {
_affected_tables.add( tab );
    }
    public
    void addAffectedColumn( lxtColumn col )
    {
_affected_columns.add( col );
    }
    public
    Iterator getAffectedTablesIterator()
    {
return _affected_tables.iterator();
    }
    public
    Iterator getAffectedColumnsIterator()
    {
return _affected_columns.iterator();
    }
    public
    double getNumRowsAffected()
    {
return _numrowsaffected;
    }
    public
    double getVanillaCost()
    {

```

```

return _vanilla;
    }
    public
    long getCount()
    {
return _count;
    }
}
// lxtTable.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
// *****
// Copyright 2002,2003 iAnywhere Solutions, Inc. All rights reserved.
// *****

package com.sybase.indexConsultant;
import java.util.*;
public class lxtTable
{
    long _id;
    String _name;
    String _creator;
    int _cluster;
    int _size;
    HashMap _physicalindexes;
    HashMap _columns;
    public lxtTable( long id, String name, String creator, int cluster, int size )
    {
_id = id;
_name = name;
_creator = creator;
_cluster = cluster;
_columns = new HashMap();
_physicalindexes = new HashMap();
_size = size;
    }
    public
    lxtColumn findColumn( long id )
    {
return (lxtColumn) _columns.get( new Long(id) );
    }
    public
    long getID()
    {
return _id;
    }
    public
    String getName()
    {
return _name;
    }
    public

```

```

    int getSize()
    {
return _size;
    }
    public
    void addColumn( IxtColumn col )
    {
_columns.put( new Long( col.getID() ), col );
    }
    public
    void addPhysicalIndex( IxtIndex ix )
    {
_physicalindexes.put( new Long( ix.getID() ), ix );
    }
    public
    IxtIndex findPhysicalIndex( long id )
    {
return (IxtIndex) _physicalindexes.get( new Long(id) );
    }
    public
    int getClusterID()
    {
return _cluster;
    }
    public
    String getCreator()
    {
return _creator;
    }
}

```